PARTIAL WRITE DATA TRACKING DURING EXPECTATION BASED EVENT VERIFICATION

Background

Designing and testing a computer architecture is an extremely complex process, involving a range of tasks from the high level such as specifying the architecture down to the low level such as determining the physical placement of transistors on a silicon substrate. Each stage of the design process requires extensive testing and verification of the design through that stage. The computer architecture is typically simulated during the design process before building and testing the hardware.

The testing process is complicated further for architectures supporting multiple cache memories. For example, a computer architecture may support multiple processors having either a shared memory, multiple dedicated memories, or both, as well as multiple cache memories (referred to hereinafter simply as caches). Multiple memory agents are also provided to handle memory operations or transactions in the system that access the shared memory or other memories and the caches. For example, one of the processors may initiate a read transaction to read a line of memory. The line of memory may be stored in one or more locations in the system, such as in the shared memory and in one or more of the caches. The memory agents work together to determine the source from which the line of memory should be read for the processor.

The memory agents and memories may be connected in a number of ways, such as by a bus or by a point to point

link network using any of a number of suitable protocols. A single memory transaction may therefore be quite complex, involving requests and data being sent back and forth among the multiple memory agents, memories and caches. The sequence of data transmissions depends upon the type of transaction (read, write, etc.), the locations and states of the line of memory in the system, the bus protocol employed, etc. Therefore, testing the operation of the memory agents in the system can be an extremely complex and data-intensive procedure.

One aspect of testing the operation of the memory agents in the system involves tracking data consistency during a partial write operation. A partial write operation allows portions of a block of memory to be written without affecting the remainder of the block of memory. For example, one or more data bits in a line of cache memory may be written without overwriting the remaining data bits in the line. A typical partial write operation involves performing a local memory read and snooping one or more other memory agents to obtain the most recent copy of the block of memory to be partially overwritten, masking off the parts of the block of memory to be protected, then performing the write operation to overwrite the unmasked portions of the block of memory.

Summary

An exemplary embodiment may comprise a computer implemented method of verifying a partial write operation in an agent, including gathering partial write data received by the agent and at least one version of existing data to be partially overwritten by the partial write data, determining a most recent version of the

existing data to be partially overwritten, and partially overwriting the most recent version with the partial write data to form expected data. When the agent performs the partial write operation, the actual data is compared with the expected data.

Another exemplary embodiment may comprise an apparatus for tracking data for a partial write operation during testing of a memory agent, the apparatus including at least one computer readable medium having computer readable program code stored thereon. The computer readable program code includes program code for obtaining partial write data and an associated bitmask from an initial partial write request, program code for obtaining a freshest copy of data to be overwritten by the partial write operation, program code for generating expected data based on the freshest copy of data, the partial write data and the bitmask, and program code for comparing actual data generated by the memory agent for the partial write operation with the expected data.

Another exemplary embodiment may comprise an apparatus for testing the operation of a memory agent during a partial write operation. The apparatus includes means for generating expected data as a prediction of data to be generated by the memory agent during the partial write operation, and means for comparing the expected data with the data generated by the memory agent during the partial write operation.

Brief Description of the Drawings

Illustrative embodiments are shown in the accompanying drawings, in which:

FIG. 1 is a block diagram of an exemplary group of memory agents in a computer system and an exemplary memory operation in the group;

FIG. 2 is a flow chart of an exemplary operation for gathering data for use in tracking data consistency in partial write operations in an expectation-based memory agent checker; and

FIG. 3 is a flow chart of an exemplary operation for generating expected data for a partial write operation in an expectation-based memory agent checker.

Description

The drawing and description, in general, disclose a method and apparatus for tracking data consistency in partial write operations in an expectation-based memory agent checker. The checker verifies proper operation of a memory agent in a computer system, and may operate as described in U.S. Patent Application Serial No. _____, entitled "EXPECTATION BASED EVENT VERIFICATION" (Attorney Docket No. 200207608-1) filed concurrently herewith, which is incorporated herein by reference for all that it discloses. The checker tests the memory agent by monitoring the inputs to the memory agent and generating expectations for events which should occur at the outputs of the memory agent. This enables black-box style testing of the agent, without requiring that the internal state of the agent be probed. The inputs and outputs of the agent are monitored, and as portions of a memory transaction are received by the agent, certain events may be expected to be transmitted from the agent in response.

Transaction records are established by the checker

to hold data relating to a transaction, and expectation records are established to hold data relating to an expected event generated by the agent. Transaction records and their associated expectation records are stored in data structures. During architectural testing of a system in which the agent is located, the operation of the agent may be tested simply by monitoring the inputs and outputs (I/O) of the agent and using transaction records and associated expectation records to verify that the agent is handling transactions properly.

Data consistency during partial write operations or transactions may be tracked in an exemplary embodiment by monitoring incoming messages at the inputs of the agent to determine what data should be written. Examples of incoming messages to be considered include the partial data to be written and the bitmask (also referred to as "byte enables") indicating what portions of a block of memory should be overwritten. Other examples of incoming messages to consider include previous copies of the data in the block of memory from various locations in the system and in various states, such as unmodified (clean) or modified (dirty). The checker uses this information to determine what data should be written by the agent to properly perform the partial write operation, as will be described in detail below.

The exemplary agent to be tested assembles data for partial write operations from various sources, and the checker verifies that the agent performs properly. The agent assembles the data for a partial write operation by receiving the partial data to be written and a bitmask indicating what portions of a block of memory are to be overwritten. The agent then searches the system for all source copies of the block of memory to obtain the most recent copy for use as a base for the partial write

operation. The agent applies the partial data to the base data to form a complete block of memory that has been partially overwritten as requested, and stores the block of memory in a standard write operation to one or more storage locations in the system. (The exemplary embodiment to be described herein is directed at verifying local partial write operations in a memory agent, wherein the agent writes the partial data to its local cache memory.) The checker monitors the request and data inputs to the agent and verifies that the appropriate data is written by the agent, as will be described in detail herein.

Before continuing with the detailed description of tracking data consistency during partial write operations, the exemplary checker will be described. However, it should be noted that the method and apparatus for tracking data consistency during partial write operations is not limited to use in the exemplary checker described herein, but may be adapted for use in any expectation-based checker for verifying the operation of a memory agent in a computer system. For example, data may be gathered and stored in any suitable manner.

The term "agent" is used herein to refer to any component of a system that receives incoming signals or transactions and generates outgoing signals or events as a result, wherein expected values or states of the outgoing signals may be determined based on the incoming transactions, possibly in connection with other information, before the actual outgoing signals are generated by the agent. In one exemplary embodiment, the agent comprises a memory agent in a computer system.

A transaction, as the term is used herein, corresponds to an entire memory operation, and starts with a request from an originating agent to perform the

memory operation. The transaction may also include snoops, read requests, responses, and writes, etc., for a partial write operation. The transaction ends when the memory operation is complete. In one exemplary embodiment, this is indicated when the originating agent sends a completion message.

The testing tool, or checker, for monitoring the I/O of the agent and for generating and using the transaction records and expectation records may be implemented in any suitable manner, such as in a software application or in an electronic circuit. In one exemplary embodiment, the checker is written as a software application using the C++ programming language. The checker may be used to test agents during architectural verification at any stage, including in a software simulation of an architecture and a test of actual hardware. The transaction records and expectation records may be stored in any data structure suitable for maintaining the relationship described herein. In the exemplary embodiment of the checker, transaction records are stored in a transaction list vector. The vector is a container template, defined in the C++ Standard Template Library (STL), that resembles a C++ array in that it holds zero or more objects of the same type. The vector container is defined as a template class, meaning that it can be customized to hold objects of any type. Each of these objects may be accessed individually, using an iterator to step through the vector. The transaction list vector is dynamically expandable by adding new transaction records to the end of the vector. Each transaction record contains an expectation list vector for storing expectation records associated with that transaction record.

When a new transaction is detected at the input of

the agent, a transaction record is created and is added to the transaction list vector. When an event expectation is generated, that is, when enough information is available to determine that an event should be generated by the agent, an expectation record is added to the expectation list vector for the appropriate transaction record. For example, if the agent receives a request to partially write a line of memory in a region belonging to that agent, the checker may expect to see a local read request sent from the agent to a cache connected to the agent to obtain the line of memory before partially overwriting the line. In this example, a transaction record would be established for the partial write transaction, and an expectation record would be added to the expectation list vector in that transaction record, indicating that a local read request should be sent from the agent to that cache.

The term "stimulus" is used herein to refer generally to any input to an agent under test for which an output from the agent under test may be expected by the checker, that is, for which an expectation may be generated by the checker. This includes, for example, an initial request to perform a partial write operation and the responses to snoops by the agent under test. The term "event" is used herein to refer to any output from the agent under test, which generally should be triggered by a "stimulus". Thus, the "stimulus" is the input to the agent under test which triggers an outgoing "event".

When an event takes place, that is, when the checker detects an outgoing event from the agent, the transaction is identified, and the checker searches the transaction list vector to find the appropriate transaction record. The transaction may be identified in any suitable manner. In many instances, the event may contain a transaction

identification (ID). In other instances, if the event does not contain a transaction ID, the checker identifies the transaction using other information in the system such as an indication of what transaction is current, or an indication in the event of what line of memory is being accessed. For example, if the transaction ID is not directly available from an outgoing event, the checker may use a C++ STL map to look up transaction ID's from information contained in the outgoing event or elsewhere.

If no transaction record is found for the outgoing event in the transaction list vector, the checker signals an error, indicating that an event has occurred without having been triggered by a stimulus.

If the transaction record is found in the transaction list vector, the expectation list vector is traversed, looking for an expectation record matching the detected event. If none is found, the checker signals an error, indicating that an unexpected event has occurred. If a match is found, the match may be logged and the expectation record may be deleted from the expectation list vector.

An exemplary system including a memory agent being tested, and a partial write memory operation in that system, will now be described. Referring now to FIG. 1, a group of agents 10, 12, 14 and 16 in a computer system is shown. The agents 10, 12, 14 and 16 may be interconnected by a bus (not shown) or other type of interface. The checker described herein may be used to test an agent connected to other devices in a system by any bus, interface, or protocol or by any combination of buses, interfaces, or protocols in which an agent receives stimuli and transmits events based on these stimuli. In the exemplary embodiment, the agents 10, 12,

14 and 16 are interconnected by a point-to-point (P2P) link network.

A P2P link network is a switch-based network which may have one or more crossbars (not shown) acting as switches between components such as memory agents, memories, processor cores, or other devices. Messages in transactions are directed to specific components and are routed appropriately in the P2P link network by crossbars. This reduces the load on components because they don't need to examine each broadcast block of information as they would if connected by a bus. Messages in the P2P link network need not occur in any specific order. Transactions on the P2P link network are packet-based, with each packet containing a header with routing and other information. Packets containing requests, responses, and data are multiplexed, so portions of various transactions may be interspersed with many others in time. Transmissions are length-limited, with each length-limited block of data called a flit. Thus, a long packet will be broken into several flits, and transactions typically require multiple packets.

During the exemplary partial write memory operation, the Originating Agent 10 sends partial data and one or more byte enables 20 to an Agent Under Test 12. The partial data is intended to be written over a specified block of memory, such as over a line of memory, part of a line, or a group of lines, etc. The byte enables, or bitmask, indicate what portions of the block of memory are to be overwritten by the partial data.

When a partial write operation is carried out through the Agent Under Test 12, it may be expected that the Agent Under Test 12 will store the resulting data in its local memory space 24, such as a cache. The method and apparatus for tracking data consistency during

partial write operations therefore monitors the inputs to the Agent Under Test 12 to generate an expectation of an event writing data to the local memory space 24. The expectation contains the appropriate data to be written, based on the partial data and byte enables in the partial write request, and on existing copies of the memory to be partially overwritten, either in modified form, if available, or unmodified form.

The Originating Agent 10 may comprise, for example, a processor that is attempting to overwrite part of a block of memory such as several bits in a line of memory. The Agent Under Test 12 queries any other devices in the system that may have a copy of the line of memory in order to obtain the most recent version to use as the source data to be partially overwritten. In this exemplary operation, the Agent Under Test 12 transmits a local memory read message 22 to the connected memory space 24, requesting the line of memory, and the memory space 24 returns the line of memory 26 to the Agent Under Test 12, where it is temporarily stored internally. In this example, a clean, or unmodified, copy of the line of memory was stored in local memory space 24. The Agent Under Test 12 concurrently sends snoops 30 and 32 to other agents 14 and 16, respectively, in the system, requesting the line of memory from them. Agent A 14 replies with an "Invalid" message 34, indicating that the line of memory is not stored in Agent A 14 or its associated memory space (not shown). In this example, the line of memory was stored in modified form in the memory space (not shown) associated with Agent B 16. Therefore, Agent B 16 obtains the modified copy of the line of memory, either from its internal cache or in its associated memory space (not shown), and transmits the modified line of memory 36 to the Agent Under Test 12.

The Agent Under Test 12 receives the modified line of memory 36 and overwrites its internal copy of the line of memory with the received modified line 36. The Agent Under Test 12 then partially overwrites the internal copy of the modified line with the data in the original partial write request 20, using the byte enables in that request 20. Finally, the Agent Under Test 12 writes the partially overwritten, modified, line of memory 40 to the local memory space 24.

Other messages (not shown) may be sent throughout the system during the memory operation, both related and unrelated to the memory operation, such as a completion message from the Originating Agent 10 to the Agent Under Test 12. Note that the sequence of transactions and events described for this memory operation are purely exemplary. The actual transactions and events involved in a memory operation are dependent on the configuration of the system, the types of interfaces between devices in the system, the locations and states of the line of memory to be partially overwritten, etc.

In this exemplary operation, the Agent Under Test 12 receives (and the checker copies) three incoming stimuli containing data to be considered when generating data for an expectation for the final write event 40. First, the original partial write request 20 contains partial data to be written. Second, the clean data 26 from the local memory space 24 is temporarily stored to use as the base for the partial write operation if no modified data is returned as a result of a snoop (e.g., 30 or 32). Finally, the modified data 36 replaces the clean data 26 to use as the base for the partial write operation. The checker considers all three of these data sources, finally using the modified data 36 as the base to be partially overwritten by the partial data in the original

partial write request 20 using the byte enables in the original partial write request 20.

For the present discussion, only these three stimuli, the original request 20 from the Originating Agent 10 to the Agent Under Test 12, the clean line of memory 26 from the local memory space 24, and the modified line of memory 36 from Agent B 16, will be considered. Although other stimuli may arrive at the Agent Under Test 12 during a partial write operation, in this example these are the only three stimuli containing data to be considered when formulating the data for the expectation of the final write event. Furthermore, although multiple events may be expected based on these three stimuli, as described in "EXPECTATION BASED EVENT VERIFICATION" which was incorporated herein above, only the expectation of the final write event will be considered herein. Other possible events will not be considered. For example, based on the original request 20, expectations may be generated by the checker for the local read event 22 and for each of the snoop events 30 and 32. However, the method and apparatus for tracking data consistency during partial write operations is concerned only with gathering the data to be placed in the expectation for the final write event, so these other expectations will not be discussed herein.

In the first stimulus, when the checker detects the original request 20 from the Originating Agent 10 at an input to the Agent Under Test 12, the checker determines that the transaction is new and adds a transaction record to the transaction list vector. The checker also generates an expectation for a local write event (the write event 40 to the local memory space 24 or cache of the Agent Under Test 12) and adds the associated expectation record to the transaction record. The

expectation record contains data structures for holding the partial data to write, the byte enables, and the base data (both clean and modified) to be partially overwritten by the partial data using the byte enables. The checker gathers the partial data to write and the byte enables from the original request 20 and stores them in the expectation record.

In the second stimulus, when the checker detects the clean data 26 from the local memory space 24 in response to the local read event 22, the checker copies the clean data into the expectation record. Because the expectation is for a partial write event, the clean data may or may not be used as the base for the partial write. If no modified data exists, then the clean data will be used as the base. However, if a modified version of the data does exist, it is a more recent version of the data and should be used in place of the clean data. The clean data in the second stimulus is therefore stored in a unique location in the expectation record so that it can be identified by the checker later as clean data.

In the third stimulus, when the checker detects the modified data 36 from Agent B 16, the checker collects the modified data and stores it in the expectation record. As described above, the checker will use the modified data in this case as the base for the partial write, rather than the clean data. The modified data is stored in a unique location in the expectation record and is identified as modified data.

Alternatively, incoming data may be stored in a single location, such as a data vector, in the expectation record, along with an indication of the state of the data. For example, if the checker had already received and stored modified data in a snoop response, and then detected incoming clean data from a local read,

the clean data would be discarded in this alternative embodiment rather than overwriting the stored modified data. If, on the other hand, the clean data arrived first, any modified data arriving later would overwrite the clean data in the expectation record.

Note that in the exemplary memory protocol, there should only be one copy of a line of memory in a system that has been modified, thus preventing version control complications. However, the method and apparatus for tracking data consistency during partial write operations may be adapted to any protocol now known or that may be developed in the future, including a protocol in which multiple modified copies of a line of memory exist simultaneously. The checker need only be able to assemble the appropriate line of memory for the partial write, based on the data received by the agent, and dependent on the particular protocol being used by the system.

When the checker determines that all data has arrived from any potential data sources, the data for the partial write can be assembled. In one exemplary embodiment, the checker determines that all data has arrived when either of two conditions have been satisfied: First, if modified data has been received, and second, if all responses to snoop events have been received, no modified data has been received, and a local read has completed. (Of course, the checker must also have received the partial data to write and the byte enables from the initial request for a partial write operation.) Regarding the first condition, in the exemplary embodiment discussed herein only one copy of modified data may exist in the system at any given time, so if modified data has been received, no different modified data exists and the received modified data will

be used as the base for the partial write whether or not clean data has been received. Therefore, if modified data has been received, the checker may prepare the expected data for the partial write based on the modified data. Regarding the second condition, if all snoop responses have been received and none contained modified data, the checker will base the partial write on data from the local memory space 24, returned in response to the local read operation. This data from the local read may be either clean or modified.

Note that these two conditions are based on the protocol and configuration of the exemplary memory system described herein. The conditions for determining when the checker has received enough data to formulate the expected partial write will vary depending on the protocol and configuration. An alternative protocol was mentioned above. The configuration of the memory system also affects the conditions, for example, if the agent does not generate one local read event. If the agent does not have a local memory space 24, or has direct access to multiple local memories, the exemplary conditions given above should be adapted to reflect the system configuration.

The exemplary checker tests these conditions at various times, including when a local read has completed, when the end of an incoming data packet is seen, and when all expected snoop responses have been received. Note again that the exemplary checker is designed to test only a single agent (e.g., the Agent Under Test 12), and therefore does not normally track and verify responses from other sources to messages from the agent being tested. However, the exemplary checker does in this case track responses from other sources to snoop messages to determine when the data for a partial write can be

assembled.

In an alternative embodiment, the checker may test these conditions to determine if the data for a partial write can be assembled when different system states are detected, depending on the configuration of the system. In another alternative embodiment, the checker may test these conditions at regular intervals.

Once one of the conditions has been satisfied, the checker may assemble the expected data for the partial write event. The checker identifies the appropriate data to use as the base for the partial write, in this case selecting the modified data stored in the expectation record over the clean data stored in the expectation record. In the exemplary embodiment, the expectation record contains storage locations for clean data, modified data, and for standard or expected data. The checker copies the selected modified data into the storage location for expected data for use as the base of the partial write operation. The checker then applies the mask to the base data and copies the partial write data over the base data, overwriting the unmasked portions of the base data to produce the expected data for the partial write operation.

The checker may then begin watching for the partial write event at the outputs of the agent, and when the event is detected, the checker compares the outgoing data in the partial write event with the expected data in the expectation record for the partial write event. If the data does not match, the checker may signal an error in the partial write event.

The following exemplary pseudo-code illustrates an exemplary embodiment for tracking data consistency in partial write operations. Note that the following pseudo-code is not a single sequential routine, but

contains multiple pieces of code that are triggered based on various circumstances, such as the detection of the last incoming data flit of a stimulus or the detection of the first outgoing data flit of an event.

When the checker detects the initial request, check whether a local write should be performed when a third party sends modified data, and if so, check request type:

```
if type is partial write{
   pass partial data variable pointer to data event watch
list
   store byte enables in expectation}
else // not partial write
   handle normally (pass regular data location to watch
list)
```

When read data comes back as a result of a local read request, check the initial request type:

```
if the initial request was any type of partial write{
   divert data into a special read_data vector of
affiliated local write event
   if read_data.size == 16 // ensure data is complete
      ProcessPartialWrites(transaction id);}
else // request not for partial write
   handle normally (find affiliated final write event and
put data in standard data location)
```

When a snoop response comes back Modified (dirty) and includes a fresher copy of the desired data, check the initial request type:

```
if the initial request was any type of partial write{
```

```
   An associated local write expectation already exists,
so don't need to add one.

   Instead, set up a watchlist entry, and point to
variable modified_data in the pre-existing expectation.
Later, when the watchlist is in use, the data will
automatically be channeled into this new location.}
else // request not for partial write
   handle normally

ProcessPartialWrites function:

Loop through all expectations associated with passed-in
transaction id
If (expectation's modified_data.size == 16)
   OR
   ((transaction's snoop response count == transaction's
expected snoop count) AND (no modified responses
received) AND (expectation's read_data.size == 16)){
   if modified_data.size != 16
      copy read_data contents to standard data location
   else
      copy modified_data contents to standard data
location

   assert(expectation's standard data location holds 16
elements)

   mask = initial request's partial write length setting
   if(initial request's address & mask)
      Log error, address not properly boundary-aligned

   /* copy over mask-included zeros */
   Expectation's standard data value[0] &=
```

```
        ~(~partial_data[0] & byte enable mask)

      /* copy over mask-included ones */
      Expectation's standard data value[0] |=
        (partial_data[0] & byte enable mask)

    Reset current address and address counters for
upcoming write check
    }
```

ProcessPartialWrites is also called when all expected snoop responses have been received (i.e., when the transaction's snoop response count equals the expected snoop count), and when the end of an incoming data packet is seen.

Several portions of the pseudo-code above include setting a pointer in a watch list to a data variable (e.g., a data vector) in an expectation. A watch list is used to collect data from incoming flits and to copy data to the appropriate storage location in an expectation. The watch list contains an entry for each stimulus for which the agent is waiting for data. For example, if a header flit has been sent to the agent as a response to a read request, the header may contain some data and may indicate that more data is on the way, for example if the response is sending a modified line of memory to the agent under test. An entry to the watch list indicates that the agent will be receiving data which should be placed in the expectation record.

When the header flit is first seen, an entry is made to the watch list, the entry including the correlative information that will appear in each incoming data flit, such as the address of the memory line and the identity

of the source. The entry also includes a pointer to the data vector of the transaction record's expectation record to which the incoming data flit belongs. As each incoming flit containing data is received (which may include the header), the correlative information is read from the incoming flit, and the watch list is searched for an entry containing that same correlative information, for example, the same source ID and memory address. If multiple transactions are taking place in the system simultaneously, multiple watch lists may exist, so the checker would need to look through each entry of several watch lists until the correct entry is located.

When the appropriate watch list entry is located, the data in the incoming flit is copied to the location to which the watch list entry pointer is directed, that is, into the appropriate data vector in the expectation record for the stimulus or for the final write event.

The first three sections of pseudo-code above handle three different stimuli, an initial request, a response to a local read, and a response to a snoop with modified data. In the system for which the exemplary pseudo-code above was designed, data stored in the local memory space (e.g., 24) for an agent being tested (e.g., 12) is always in the clean (unmodified) state, and is always available. Therefore, data from snoop responses is only used when modified, because clean data can always be retrieved from the local memory space (e.g., 24). The pseudo-code may be adapted for use with other configurations. For example, in an alternative embodiment, clean data is not always available from the local memory space and may be obtained from snoop responses. In another alternative embodiment, modified data may be obtained from local reads. To adapt to these types of alternatives, the

pseudo-code would determine whether incoming data was clean or dirty, regardless of the source, and would store the data accordingly in the appropriate expectation record.

The ProcessPartialWrites function first tests the two conditions described above to determine if the proper base data has been received from which to assemble the expected data for the partial write. The pseudo-code above determines whether a particular type of data has been received by checking the size of the data - in the exemplary embodiment, the size of a complete block of data is 16. (Units are not important, this is simply a description of how the exemplary pseudo-code above determines whether a particular type of data has been completely received. Alternative methods may be employed for determining this information.)

The exemplary pseudo-code employs a two-step process for overwriting the base data with the partial write data, first copying the 0's from the partial write data over the base data, then copying the 1's from the partial data over the base data. The byte enable (bitmask) contains 1's in all bit locations to be overwritten in the base data, and 0's in bit locations to be protected or masked. To copy the partial write data 0's over the base data, a bitwise inversion of the partial write data (partial_data[0]) is combined with the bitmask in a bitwise AND operation, and the combination is bitwise inverted. The result has 0's only in bit locations where the partial write data had a 0 and the bitmask had a 1, with 1's in all other bit locations of the result. The base data (which was copied into the expectation's standard data location) is combined with the result in a bitwise AND operation, so that the bit locations in the base data where the partial write data was 0 are set to

0, and all other locations are left unchanged. To copy the partial write data 1's over the base data, the partial write data (partial_data[0]) is combined with the bitmask in a bitwise AND operation. The result has 1's only in bit locations where the partial write data and the bitmask both had a 1, with 0's in all other bit locations of the result. The base data is combined with the result in a bitwise OR operation, so that he bit locations in the base data where the partial write data was 1 are set to 1, and all other locations are left unchanged. Note that the method and apparatus for tracking data consistency in partial write operations is not limited for use with any particular method of partially overwriting the base data and may use an alternative method from that described above.

Once the expected data for the partial write operation has been assembled and stored in the expectation record for the final write event, the checker may monitor the outputs of the agent for the partial write event. When the partial write event is detected, the checker may compare the actual outgoing data with the expected data, and may signal an error if the data does not match.

An exemplary operation for gathering data for use in tracking data consistency in partial write operations is summarized in the flowchart of FIG. 2. The checker begins assembling data for an expectation of a partial write operation when an incoming stimulus is received 50. If 52 the stimulus is an initial request for a partial data write, the checker gathers 54 the partial data to write and the bitmask indicating what memory positions to overwrite from the initial request, and stores the partial data and bitmask in an expectation record for the final write event. If 60 the incoming stimulus is a

response to a local read generated by the agent being tested, the checker gathers 62 the data in the response and stores it in the expectation record. If 64 the incoming stimulus is a modified response to a snoop, the checker gathers 66 the modified data in the response and stores it in the expectation record. As discussed above, the clean data and modified data may be stored separately in the expectation record for later use in identifying the most recent or fresh copy of the data.

An exemplary operation for assembling the gathered data into expected data for the partial write event is summarized in the flowchart of FIG. 3. The checker determines 70 whether enough data has been received to use as the base for the partial write, as described above. For example, the checker may proceed once a modified response has been fully received 66. If the checker has not received enough data to assemble the expected data, the checker continues monitoring the inputs of the agent to collect more data. If enough data has been received to use as a base, the checker identifies 72 the most fresh copy of the data to use as the base for the partial write. The checker then partially overwrites 74 the base data with the partial data using the bitmask. Finally, the checker stores 76 the resulting block of memory in the expectation record for the final write event. The checker may then begin watching the outputs of the agent for the final write event, and may then compare the stored expected data with the actual outgoing data in the final write event. If the data does not match, the checker may signal an error.

Various computer readable or executable code or electronically executable instructions have been referred to herein. These may be implemented in any suitable manner, such as software, firmware, hard-wired electronic

circuits, or as the programming in a gate array, etc. Software may be programmed in any programming language, such as machine language, assembly language, or high-level languages such as C or C++. The computer programs may be interpreted or compiled.

Computer readable or executable code or electronically executable instructions may be tangibly embodied on any computer-readable storage medium or in any electronic circuitry for use by or in connection with any instruction-executing device, such as a general purpose processor, software emulator, application-specific circuit, a circuit made of logic gates, etc. that can access or embody, and execute, the code or instructions.

Methods described and claimed herein may be performed by the execution of computer readable or executable code or electronically executable instructions, tangibly embodied on any computer-readable storage medium or in any electronic circuitry as described above.

A storage medium for tangibly embodying computer readable or executable code or electronically executable instructions includes any means that can store, transmit, communicate, or in any way propagate the code or instructions for use by or in connection with the instruction-executing device. For example, the storage medium may include (but is not limited to) any electronic, magnetic, optical, or other storage device, or any transmission medium such as an electrical conductor, an electromagnetic, optical, infrared transmission, etc. The storage medium may even comprise an electronic circuit, with the code or instructions represented by the design of the electronic circuit. Specific examples include magnetic or optical disks, both

fixed and removable, semiconductor memory devices such as memory cards and read-only memories (ROMs), including programmable and erasable ROMs, non-volatile memories (NVMs), optical fibers, etc. Storage media for tangibly embodying code or instructions also include printed media such as computer printouts on paper which may be optically scanned to retrieve the code or instructions, which may in turn be parsed, compiled, assembled, stored and executed by an instruction-executing device. The code or instructions may also be tangibly embodied as an electrical signal in a transmission medium such as the Internet or other types of networks, both wired and wireless.

While illustrative embodiments have been described in detail herein, it is to be understood that the concepts disclosed herein may be otherwise variously embodied and employed, and that the appended claims are intended to be construed to include such variations, except as limited by the prior art.

WHAT IS CLAIMED IS:

1. A computer implemented method of verifying a partial write operation in an agent, said method comprising:

gathering partial write data received by said agent;

gathering at least one version of existing data to be partially overwritten by said partial write data, said at least one version of said existing data being received by said agent;

determining a most recent version among said at least one version of said existing data to partially overwrite with said partial write data; and

partially overwriting said most recent version of said existing data with said partial write data.

2. The method of claim 1, further comprising gathering a bitmask received by said agent.

3. The method of claim 1, wherein said partial write data is gathered at an input of said agent.

4. The method of claim 1, further comprising:

gathering output data generated by said agent for said partial write operation; and

comparing said partially overwritten most recent version with said output data.

5. The method of claim 4, further comprising signaling an error if said partially overwritten most recent version does not match said output data.

6. The method of claim 1, wherein said at least one version of existing data comprises a modified version of

said existing data and an unmodified version of said existing data.

7. The method of claim 6, wherein said determining said most recent version among said at least one version of said existing data comprises determining that said modified version is more recent than said unmodified version.

8. The method of claim 1, wherein said at least one version of existing data comprises a plurality of versions of said existing data, and wherein said plurality of versions of said existing data have a same memory address and are obtained from different sources.

9. An apparatus for tracking data for a partial write operation during testing of a memory agent; comprising:

a. at least one computer readable medium; and

b. computer readable program code stored on said at least one computer readable medium, said computer readable program code comprising:

i. program code for obtaining partial write data and an associated bitmask from an initial partial write request;

ii. program code for obtaining a freshest copy of data to be overwritten by said partial write operation;

iii. program code for generating expected data based on said freshest copy of data, said partial write data and said bitmask; and

iv. program code for comparing actual data generated by said memory agent for said partial write operation with said expected data.

10. The apparatus of claim 9, wherein said program code for obtaining said partial write data and said associated bitmask comprises:

program code for monitoring at least one input of said memory agent for a partial write request containing said partial write data and said associated bitmask; and

program code for storing said partial write data and said associated bitmask.

11. The apparatus of claim 10, wherein said program code for storing said partial write data and said associated bitmask comprises code for storing said partial write data and said associated bitmask in an expectation record for a final partial write event.

12. The apparatus of claim 9, wherein said program code for obtaining said freshest copy of data comprises:

program code for monitoring at least one input of said memory agent for incoming data having an address at least partially overlapping an address of said partial write data; and

program code for storing said incoming data.

13. The apparatus of claim 12, wherein said program code for monitoring said least one input of said memory agent for said incoming data comprises monitoring for clean and modified versions of data to be partially overwritten by said partial write data.

14. The apparatus of claim 13, wherein said program code for storing said incoming data comprises code for storing said clean version of data and said modified version of

data separately.

15. The apparatus of claim 14, wherein said program code for generating said expected data based on said freshest copy of data, said partial write data and said bitmask comprises partially overwriting said modified version of said data if said modified version is received and partially overwriting said clean version of said data if said modified version is not received.

16. The apparatus of claim 14, wherein said program code for storing said incoming data comprises code for storing said clean version of data and said modified version of data in an expectation record for a final partial write event.

17. The apparatus of claim 9, wherein said program code for generating said expected data based on said freshest copy of data, said partial write data and said bitmask comprises program code for applying said bitmask to said freshest copy of data to mask off protected portions of said freshest copy and for copying said partial write data over unmasked portions of said freshest copy.

18. The apparatus of claim 9, wherein said program code for generating said expected data based on said freshest copy of data, said partial write data and said bitmask comprises program code for determining whether a freshest possible copy of said data has been obtained before generating said expected data.

19. The apparatus of claim 18, wherein said program code for determining whether said freshest possible copy of said data has been obtained comprises program code for

determining whether modified data has been obtained at an input of said memory agent, said modified data being used as said freshest possible copy to be partially overwritten by said partial data.

20. The apparatus of claim 18, wherein said program code for determining whether said freshest possible copy of said data has been obtained comprises program code for determining that all expected data has been received at an input of said memory agent, said expected data including unmodified data, and for determining that no modified data has been received, said unmodified data being used as said freshest possible copy to be partially overwritten by said partial data.

21. The apparatus of claim 20, wherein said program code for determining that all expected data has been received at an input of said memory agent comprises program code for determining that a local read response containing said unmodified data has been received at said input of said memory agent.

22. An apparatus for testing the operation of a memory agent during a partial write operation, comprising:

means for generating expected data as a prediction of data to be generated by said memory agent during said partial write operation; and

means for comparing said expected data with said data generated by said memory agent during said partial write operation.

Abstract

A computer implemented method of verifying a partial write operation in an agent includes gathering partial write data received by the agent and at least one version of existing data to be partially overwritten by the partial write data, determining a most recent version of the existing data to be partially overwritten, and partially overwriting the most recent version with the partial write data to form expected data. When the agent performs the partial write operation, the actual data is compared with the expected data.

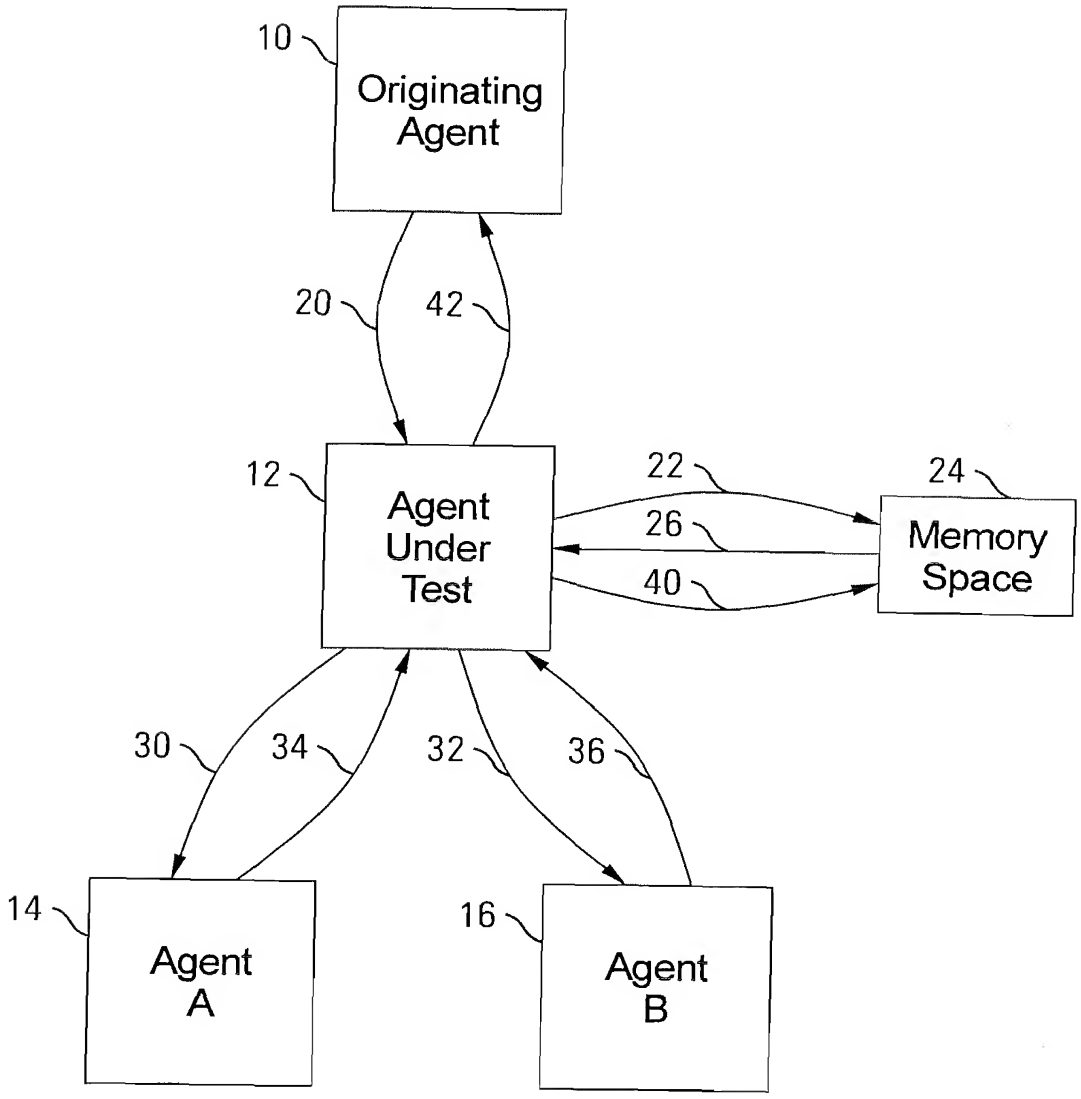
10
Originating Agent
20
42
12
Agent Under Test
22
26
40
24
Memory Space
30
34
32
36
14
Agent A
16
Agent B

FIG. 1

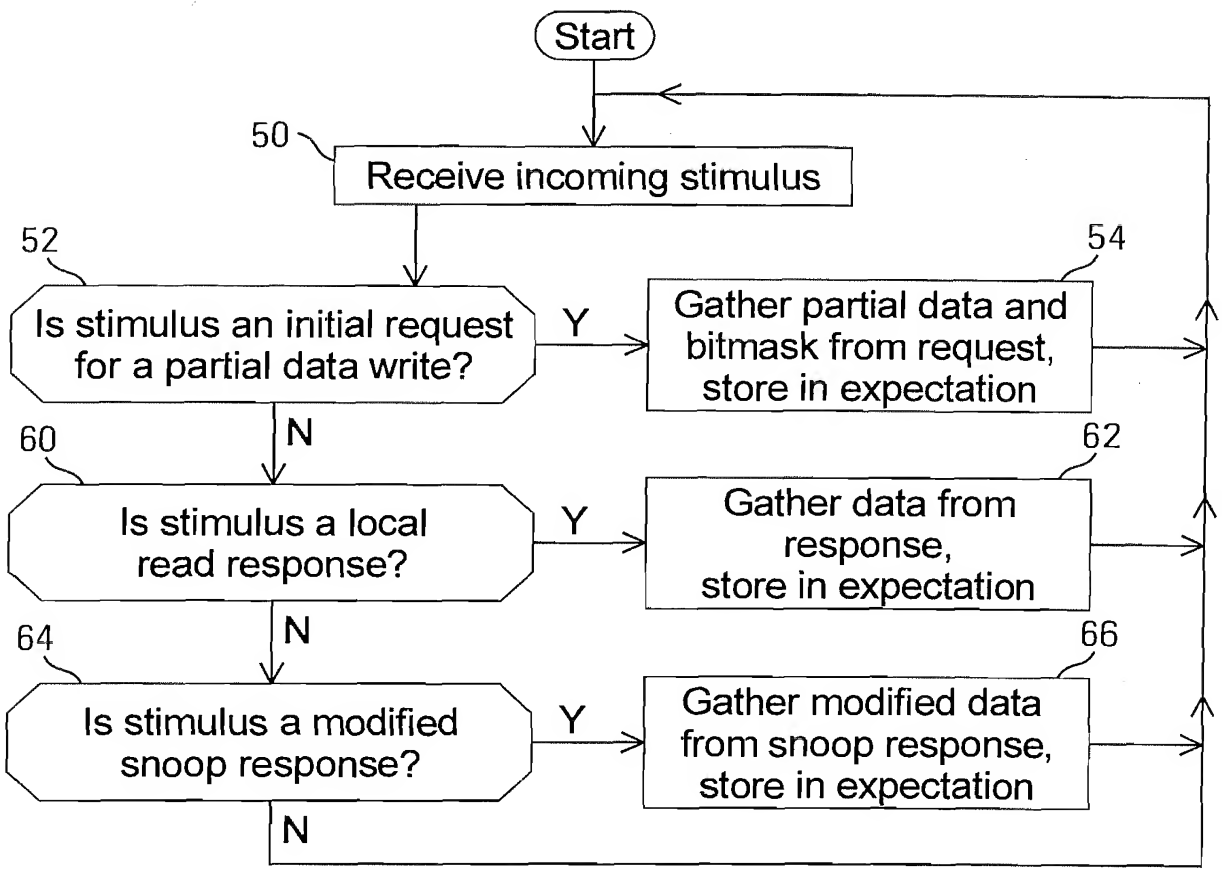
Start
50
Receive incoming stimulus
52
Is stimulus an initial request for a partial data write?
Y
54
Gather partial data and bitmask from request, store in expectation
N
60
Is stimulus a local read response?
Y
62
Gather data from response, store in expectation
N
64
Is stimulus a modified snoop response?
Y
66
Gather modified data from snoop response, store in expectation
N

FIG. 2

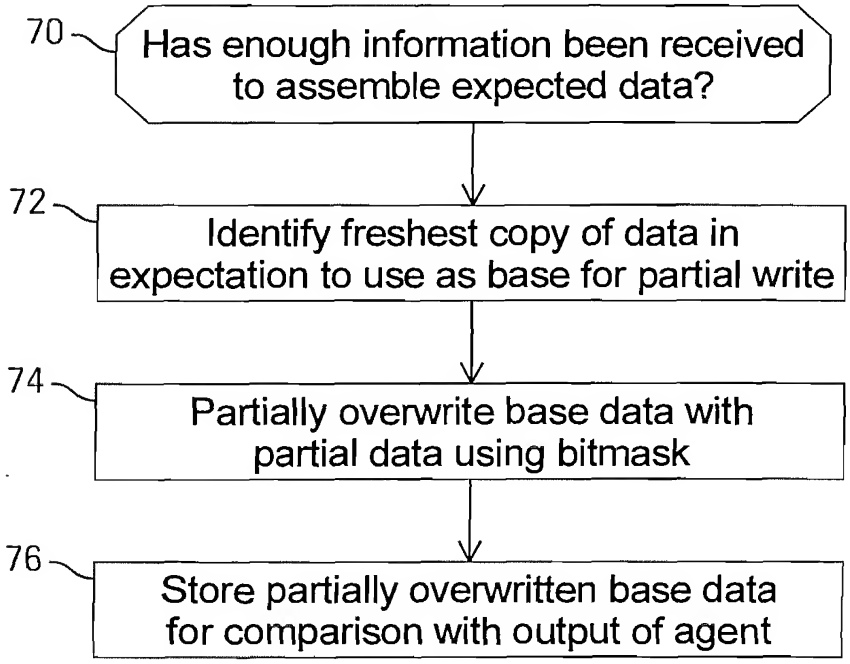
70
Has enough information been received to assemble expected data?
72
Identify freshest copy of data in expectation to use as base for partial write
74
Partially overwrite base data with partial data using bitmask
76
Store partially overwritten base data for comparison with output of agent

FIG. 3